Joshua Shoup

Version 1.1

# Table of Contents

Hierarchical Index 4

Class Hierarchy 4

Class Index 5

Class List 5

Class Documentation 6

TigerStop\_API.TigerStop\_IO 6

Public Member Functions 6

Static Public Member Functions 8

Public Attributes 8

Properties 8

Private Member Functions 8

Private Attributes 8

Detailed Description 9

Member Function Documentation 9

TigerStop\_API.TigerStop\_Com 40

Classes 40

Public Member Functions 40

Public Attributes 40

Protected Member Functions 40

Static Protected Member Functions 41

Properties 41

Events 41

Private Member Functions 41

Private Attributes 42

Static Private Attributes 43

Detailed Description 43

Member Function Documentation 43

TigerStop\_API.TigerStop\_Com.LastAck 58

Public Member Functions 58

Properties 58

Private Attributes 58

TigerStop\_API.TigerStop\_Com.LastCommand 59

Public Member Functions 59

Properties 59

Private Attributes 59

TigerStop\_API.MessageEvent 60

Public Member Functions 60

Properties 60

Private Attributes 60

# Hierarchical Index

## Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

EventArgs

TigerStopAPI.MessageEvent 60

TigerStopAPI.TigerStop\_Com.LastAck 58

TigerStopAPI.TigerStop\_Com.LastCommand 59

TigerStopAPI.TigerStop\_Com 40

TigerStopAPI.TigerStop\_IO 6

# Class Index

## Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

**TigerStopAPI.TigerStop\_Com.LastAck**  58

**TigerStopAPI.TigerStop\_Com.LastCommand**  59

**TigerStopAPI.MessageEvent** 60

**TigerStopAPI.TigerStop\_Com**  40

**TigerStopAPI.TigerStop\_IO**  6

# Class Documentation

## TigerStopAPI.TigerStop\_IO Class Reference

Inherits **TigerStopAPI.TigerStop\_Com**.

### Public Member Functions

* **TigerStop\_IO** (string comPort, int baud)

*A basic parameterized constructor for TigerStop\_IO that sends 'comPort' and 'baud' to TigerStop\_Com() and opens a serial connection with the given 'comPort' and 'baud' and ensures that the connection.*

* bool **MoveTo** (double position)

*Sends a move command to the machine to move to the desired position waiting for the move to finish.*

* bool **MoveTo** (double position, int timeout)

*Sends a move command to the machine to move to the desired position, waiting for the move to finish or for the given timeout duration before returning.*

* bool **MoveTo** (double position, ref BackgroundWorker b)

*Sends a move command to the machine to move to the desired position.*

* bool **MoveTo** (double position, int timeout, ref BackgroundWorker b)

*Sends a move command to the machine to move to the desired position, waiting for the move to finish or for the given timeout duration before returning.*

* bool **MoveTo** (string position)

*Sends a move command to the machine to move to the desired position waiting for the move to finish.*

* bool **MoveTo** (string position, int timeout)

*Sends a move command to the machine to move to the desired position, waiting for the move to finish or for the given timeout duration before returning.*

* bool **MoveTo** (string position, ref BackgroundWorker b)

*Sends a move command to the machine to move to the desired position.*

* bool **MoveTo** (string position, int timeout, ref BackgroundWorker b)

*Sends a move command to the machine to move to the desired position, waiting for the move to finish or for the given timeout duration before returning.*

* void **HomeDevice** ()

*Runs the home routine to return the machine to the home position.*

* bool **CycleTool** ()

*Sends a cycle tool command to the machine waiting until the tool cycle is finished.*

* bool **CycleTool** (int timeout)

*Sends a cycle tool command to the machine, waiting for the tool cycle to finish or for the given timeout duration before finishing.*

* bool **CycleTool** (ref BackgroundWorker b)

*Sends a cycle tool command to the machine waiting until the tool cycle is finished.*

* bool **CycleTool** (int timeout, ref BackgroundWorker b)

*Sends a cycle tool command to the machine, waiting for the tool cycle to finish or for the given timeout duration before finishing.*

* string **GetSetting** (int settingIndex)

*Returns the value of a desired setting at 'settingIndex'.*

* string **GetSetting** (string settingName)

*Returns the value of the desired setting whose name matches 'settingName'.*

* string **GetSetting** (int settingIndex, int timeout)

*Returns the value of the desired setting at 'settingIndex', waiting for 'timeout's duration for a response.*

* string **GetSetting** (string settingName, int timeout)

*Returns the value of the setting whose name matches 'settingName', waiting for 'timeout's duration for a response.*

* void **Stop** ()

*Sends a stop command to the machine, ending any action its currently in the middle of.*

* void **EmergencyStop** ()

*Sends an emergency stop command to the machine, ending any action its currently in the middle of and disabling the machine's drive.*

* string [] **GetAnalog** ()

*Returns the analog values tracked by the amp.*

* string **GetAnalog** (int subCommand)

*Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches.*

* string **GetAnalog** (int subCommand, int timeout)

*Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.*

* string **GetAnalog** (string subCommand)

*Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches.*

* string **GetAnalog** (string subCommand, int timeout)

*Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.*

* string [] **GetLog** ()

*Returns the 20 most recent command and error log entries.*

* string [] **GetLog** (int logIndex)

*Returns the 20 most recent command and error log entries starting at 'logIndex' and going back.*

* string [] **GetLog** (string logIndex)

*Returns the 20 most recent command and error log entries starting at 'logIndex' and going back.*

* string [] **GetCounter** ()

*Returns the 25 counter values tracked by the amp.*

* string **GetCounter** (int subCommand)

*Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches.*

* string **GetCounter** (int subCommand, int timeout)

*Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.*

* string **GetCounter** (string subCommand)

*Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches.*

* string **GetCounter** (string subCommand, int timeout)

*Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.*

* double **GetPosition** ()

*Returns the current position of the machine.*

* double **GetPosition** (int timeout)

*Returns the current position of the machine, waiting for the duration of 'timeout' for a response.*

* int **GetStatus** ()

*Returns the current status of the machine.*

* int **GetStatus** (int timeout)

*Returns the current status of the machine, waiting for the duration of 'timeout' for a response.*

* void **DriveSleep** ()

*Sends the drive sleep command to turn off the amp drive.*

* void **DriveWake** ()

*Sends the wake up command to turn on the amp drive.*

* void **WriteCommand** (string command)

*Sends 'command' to the machine to execute.*

* bool **UpdateSetting** (string settingName, double newValue)

*Changes the setting value of 'settingName' to 'newValue'.*

* bool **UpdateSetting** (string settingName, string newValue)

*Changes the setting value of 'settingName' to 'newValue'.*

* bool **UpdateSetting** (int settingIndex, double newValue)

*Changes the setting value at 'settingIndex' to 'newValue'.*

* bool **UpdateSetting** (int settingIndex, string newValue)

*Changes the setting value at 'settingIndex' to 'newValue'.*

* bool **DetectToolCycle** ()

*Listens for the expected sequence of acknowledgments from the machine, making sure that the deadman off signal followed by the deadman on signal is received by the system, meaning a full tool cycle occurred.*

* bool **DetectToolCycle** (int timeout)

*Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. Each of the waiting events will wait for the duration of 'timeout' for a response.*

* bool **DetectToolCycle** (int timeout1, int timeout2)

*Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. The first waiting event will wait for the duration of 'timeout1' for a response, and the second waiting event will wait for the duration of 'timeout2' for a response.*

* bool **DetectToolCycle** (ref BackgroundWorker b)

*Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred.*

* bool **DetectToolCycle** (int timeout, ref BackgroundWorker b)

*Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. Each of the waiting events will wait for the duration of 'timeout' for a response.*

* bool **DetectToolCycle** (int timeout1, int timeout2, ref BackgroundWorker b)

*Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. The first waiting event will wait for the duration of 'timeout1' for a response, and the second waiting event will wait for the duration of 'timeout2' for a response.*

* bool **IO\_Connection**(int ioNum, bool onOff)

Turns the desired IO connection on the IO panel on or off depending on ‘onOff’.

* bool **IO\_Connection**(int ioNum, bool onOff, int timeout)  
  *Turns the desired IO connection on the IO panel on or off depending on ‘onOff’, waiting for the duration of ‘timeout’ for a response.*
* List<double> **ScanDefectedLength**()

Sends a scan command to the machine and waits until all of the UV detected marks and the material length have been determined and returns them as a ‘List’. NOTE: ‘marks’ values are converted to imperial or metric automatically based on the current ‘MMEnable’ value.

* List<double> **ScanDefectedLength**(int timeout)

Sends a scan command to the machine and waits until all of the UV detected marks and the material length have been determined, or the duration of timeout, and returns them as a ‘List’. NOTE: ‘marks’ values are converted to imperial or metric automatically based on the current ‘MMEnable’ value.

* double **RandomLengthMeasure**()

Sends a measure command to the machine and waits until the material length has been determined. NOTE: ‘length’ values will always be in imperial inches. If metric values are desired, uncomment conversion and modify as needed.

* double **RandomLengthMeasure**(int timeout)

Sends a measure command to the machine and waits until the material length has been determined or the duration of ‘timeout’. NOTE: ‘length’ values will always be in imperial inches. If metric values are desired, uncomment conversion and modify as needed.

### Static Public Member Functions

* static List< KeyValuePair< string, int > > **Connections** ()

*Searches through all available com ports and baud rates to find potential connections by asking for serial numbers from machines that might be on the other end of the connection.*

### Public Attributes

* EventHandler **IO\_Error**

### Properties

* string **AckMessage** [get, private set]
* List< string > **SettingNames** [get, private set]

### Private Member Functions

* void **Com\_MessageReceived** (object sender, EventArgs e)

*This event listens for messages from* ***TigerStop\_Com*** *that can be used by* ***TigerStop\_IO****.*

### Private Attributes

* AutoResetEvent **ackEvent** = new AutoResetEvent(false)
* AutoResetEvent **movingEvent** = new AutoResetEvent(false)
* AutoResetEvent **cyclingEvent** = new AutoResetEvent(false)
* AutoResetEvent **deadmanOffEvent** = new AutoResetEvent(false)
* AutoResetEvent **deadmanOnEvent** = new AutoResetEvent(false)
* AutoResetEvent **homingEvent** = new AutoResetEvent(false)
* AutoResetEvent **measureEvent** = new AutoResetEvent(false)
* string **ackMessage**
* List< string > **settingNames**

### Detailed Description

**TigerStop\_IO** inherits **TigerStop\_Com** and wraps its serial communication ability in simpler, easy to manage functions. This class is the main interface between outside entities and the machine, wrapping the more esoteric machine specific commands into simple functions that inform the caller if they were able to successfully complete the desired command.

### Member Function Documentation

This function list is sorted roughly, if not completely, alphabetically:

#### void TigerStopAPI.TigerStop\_IO.Com\_MessageReceived (object *sender*, EventArgs *e*)[private]

This event listens for messages from **TigerStop\_Com** that can be used by **TigerStop\_IO**.

##### Parameters:

|  |  |
| --- | --- |
| *sender* | The object that called Com\_MessageReived() to handle an event. |
| *e* | The arguments send by the sender for the event handler to use. |

#### static List<KeyValuePair<string, int> > TigerStopAPI.TigerStop\_IO.Connections ()[static]

Searches through all available com ports and baud rates to find potential connections by asking for serial numbers from machines that might be on the other end of the connection.

##### Returns:

A 'Dictionary' where a 'string' comport name key has an 'int' baud rate value.

#### Example:

// Get all of the possible connections.

List<KeyValuePair<string, int>> connections = TigerStop\_IO.FindConnections();

if (connections.Count > 0)

{

// Use the first available connection to create a new object.

TigerStop\_IO io = new TigerStop\_IO(connections[0].Value, connections[0].Key);

}

#### bool TigerStopAPI.TigerStop\_IO.CycleTool ()

Sends a cycle tool command to the machine waiting until the tool cycle is finished.

##### Returns:

A 'bool' that signals whether the cycle command was successfully completed.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// Cut the material.

io.CycleTool();

}

#### bool TigerStopAPI.TigerStop\_IO.CycleTool (int *timeout*)

Sends a cycle tool command to the machine, waiting for the tool cycle to finish or for the given timeout duration before finishing.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An 'int' that denotes the number of milliseconds to timeout on. |

##### Returns:

A 'bool' that signals whether the cycle command was successfully completed.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// The tool cycle is particularly long, give it 10000 milliseconds, 10 seconds, to complete a cycle

// before moving on.

io.CycleTool(10000);

}

#### bool TigerStopAPI.TigerStop\_IO.CycleTool (ref BackgroundWorker *b*)

Sends a cycle tool command to the machine, waiting for the tool cycle to finish or for the given timeout duration before finishing.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An 'int' that denotes the number of milliseconds to timeout on. |

##### Returns:

A 'bool' that signals whether the cycle command was successfully completed.

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// Pass the background worker to CycleTool() so that it will cancel itself when the

// background worker has a cancellation pending.

if (!io.CycleTool(ref bkgndMakeCuts))

{

break;

}

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### bool TigerStopAPI.TigerStop\_IO.CycleTool (int *timeout*, ref BackgroundWorker *b*)

Sends a cycle tool command to the machine, waiting for the tool cycle to finish or for the given timeout duration before finishing.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An 'int' that denotes the number of milliseconds to timeout on. |
| *b* | A ‘BackgroundWorker’ that is running CycleTool() that may signal an impending cancellation. |

##### Returns:

A 'bool' that signals whether the cycle command was successfully completed.

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// Pass the background worker to CycleTool() so that it will cancel itself when the

// background worker has a cancellation pending. The tool cycle is particularly long, give

// it 10000 milliseconds, 10 seconds, to complete a cycle.

if (!io.CycleTool(ref bkgndMakeCuts, 10000))

{

break;

}

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### bool TigerStopAPI.TigerStop\_IO.DetectToolCycle ()

Listens for the expected sequence of acknowledgments from the machine, making sure that the deadman off signal followed by the deadman on signal is received by the system, meaning a full tool cycle occurred.

##### Returns:

A 'bool' denoting whether a deadman off and on signal was received in sequence, defining a full tool cycle has occurred.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// The machine does not cut automatically, so the system needs to wait for signals from the

// machine that a cycle has been done manually before it knows it is safe to continue.

io.DetectToolCycle();

}

#### bool TigerStopAPI.TigerStop\_IO.DetectToolCycle (int *timeout*)

Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. Each of the waiting events will wait for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'bool' denoting whether a deadman off and on signal was received in sequence, defining a full tool cycle has occurred.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// The machine does not cut automatically, so the system needs to wait for signals from the

// machine that a cycle has been done manually before it knows it is safe to continue. Its important

// that the tool cycle happens in 10000 milliseconds, 10 seconds, otherwise break out of the loop to

// prevent further movement.

if (!io.DetectToolCycle(10000))

{

break;

}

}

#### bool TigerStopAPI.TigerStop\_IO.DetectToolCycle (int *timeout1*, int *timeout2*)

Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. The first waiting event will wait for the duration of 'timeout1' for a response, and the second waiting event will wait for the duration of 'timeout2' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *timeout1* | An 'int' representing the desired timeout value in milliseconds the first event will wait for a response. |
| *timeout2* | An 'int' representing the desired timeout value in milliseconds the second event will wait for a response. |

##### Returns:

A 'bool' denoting whether a deadman off and on signal was received in sequence, defining a full tool cycle has occurred.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// The machine does not cut automatically, so the system needs to wait for signals from the

// machine that a cycle has been done manually before it knows it is safe to continue. It's important

// that the tool cycle starts in 10000 milliseconds, 10 seconds, and finishes in 10000 milliseconds, // 10 seconds, after starting. Otherwise, break out of the loop to prevent further movement.

if (!io.DetectToolCycle(10000, 10000))

{

break;

}

}

#### bool TigerStopAPI.TigerStop\_IO.DetectToolCycle (ref BackgroundWorker *b*)

Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred.

##### Parameters:

|  |  |
| --- | --- |
| *b* | A 'BackgroundWorker' that is running **DetectToolCycle()** that may signal an impending cancellation. |

##### Returns:

A 'bool' denoting whether a deadman off and on signal was received in sequence, defining a full tool cycle has occurred.

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// Pass the background worker to DetectToolCycle() so that it will cancel itself when the

// background worker has a cancellation pending.

if (!io.DetectToolCycle(ref bkgndMakeCuts))

{

break;

}

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### bool TigerStopAPI.TigerStop\_IO.DetectToolCycle (int *timeout*, ref BackgroundWorker *b*)

Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. Each of the waiting events will wait for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |
| *b* | A 'BackgroundWorker' that is running **DetectToolCycle()** that may signal an impending cancellation. |

##### Returns:

A 'bool' denoting whether a deadman off and on signal was received in sequence, defining a full tool cycle has occurred.

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// Pass the background worker to DetectToolCycle() so that it will cancel itself when the

// background worker has a cancellation pending, or if 10000 milliseconds, 10 seconds,

// passes before the tool is cycled.

if (!io.DetectToolCycle(10000, ref bkgndMakeCuts))

{

break;

}

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### bool TigerStopAPI.TigerStop\_IO.DetectToolCycle (int *timeout1*, int *timeout2*, ref BackgroundWorker *b*)

Listens for the expected sequence of acknowledgments from the machine, make sure that the deadman off signal followed by the deadman on signal us received by the system, meaning a full tool cycle occurred. The first waiting event will wait for the duration of 'timeout1' for a response, and the second waiting event will wait for the duration of 'timeout2' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *timeout1* | An 'int' representing the desired timeout value in milliseconds the first event will wait for a response. |
| *timeout2* | An 'int' representing the desired timeout value in milliseconds the second event will wait for a response. |
| *b* | A 'BackgroundWorker' that is running **DetectToolCycle()** that may signal an impending cancellation. |

##### Returns:

A 'bool' denoting whether a deadman off and on signal was received in sequence, defining a full tool cycle has occurred.

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// Pass the background worker to DetectToolCycle() so that it will cancel itself when the

// background worker has a cancellation pending, or if 10000 milliseconds, 10 seconds,

// passes before the tool starts a cycle or 10000 milliseconds, 10 seconds, passes before

// the tool finishes a cycle.

if (!io.DetectToolCycle(10000, 10000, ref bkgndMakeCuts))

{

break;

}

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### void TigerStopAPI.TigerStop\_IO.DriveSleep ()

Sends the drive sleep command to turn off the amp drive.

#### Example:

public void RefreshDrive()

{

// Disable the drive.

DriveSleep();

// Re-enable the drive.

DriveWake();

}

#### void TigerStopAPI.TigerStop\_IO.DriveWake ()

Sends the wake up command to turn on the amp drive.

#### Example:

public void RefreshDrive()

{

// Disable the drive.

DriveSleep();

// Re-enable the drive.

DriveWake();

}

#### void TigerStopAPI.TigerStop\_IO.EmergencyStop ()

Sends an emergency stop command to the machine, ending any action its currently in the middle of and disabling the machine's drive.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// The machine does not cut automatically, so the system needs to wait for signals from the

// machine that a cycle has been done manually before it knows it is safe to continue. Its important

// that the tool cycle happens in 10000 milliseconds, 10 seconds, otherwise emergency stop so that

// the machine doesn’t try to push anything or anyone under the tool.

if (!io.DetectToolCycle(10000))

{

io.EmergencyStop();

}

}

#### string [] TigerStopAPI.TigerStop\_IO.GetAnalog ()

Returns the analog values tracked by the amp.

##### Returns:

A 'string' array of 5 entries that hold each of the 5 analog values tracked by the amp.

#### Example:

public void DisplayAnalog()

{

// Get the array of strings.

string[] printStrings = io.GetAnalog();

// Display each of the strings.

foreach (string s in printStrings)

{

Console.WriteLine(s);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetAnalog (int *subCommand*)

Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | An 'int' that represents the desired subcommand value to retrieve. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in an expected time frame.

#### Example:

// A function to get the temperature of the amplifier.

public double GetAmpTemp(bool inCelcius)

{

if (inCelcius)

{

// The machine returns the temperature in Celcius by default, just convert it.

return Conver.ToDouble(GetAnalog(4));

}

else

{

// The temperature will need to be converted from Celcius to Fahrenheit.

return (Convert.ToDouble(GetAnalog(4)) \* 1.8 + 32);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetAnalog (int *subCommand*, int *timeout*)

Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | An 'int' that represents the desired subcommand value to retrieve. |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in the expected time frame of 'timeout'.

#### Example:

// A function to get the temperature of the amplifier.

public double GetAmpTemp(bool inCelcius)

{

if (inCelcius)

{

// The machine returns the temperature in Celcius by default, just convert it.

// Communication between the system and the machine is slow, give it 5000

// milliseconds, 5 seconds, to return with a response.

return Conver.ToDouble(GetAnalog(4, 5000));

}

else

{

// The temperature will need to be converted from Celcius to Fahrenheit.

// Communication between the system and the machine is slow, give it 5000

// milliseconds, 5 seconds, to return with a response.

return (Convert.ToDouble(GetAnalog(4, 5000)) \* 1.8 + 32);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetAnalog (string *subCommand*)

Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | An 'string' that represents the desired subcommand value to retrieve. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in an expected time frame.

#### Example:

// A function to get the temperature of the amplifier.

public double GetAmpTemp(bool inCelcius)

{

if (inCelcius)

{

// The machine returns the temperature in Celcius by default, just convert it.

return Conver.ToDouble(GetAnalog(“4”));

}

else

{

// The temperature will need to be converted from Celcius to Fahrenheit.

return (Convert.ToDouble(GetAnalog(“4”)) \* 1.8 + 32);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetAnalog (string *subCommand*, int *timeout*)

Returns the analog value related to the analog subcommand, 1-5, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | An 'string' that represents the desired subcommand value to retrieve. |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in the expected time frame of 'timeout'.

#### Example:

// A function to get the temperature of the amplifier.

public double GetAmpTemp(bool inCelcius)

{

if (inCelcius)

{

// The machine returns the temperature in Celcius by default, just convert it.

// Communication between the system and the machine is slow, give it 5000

// milliseconds, 5 seconds, to return with a response.

return Conver.ToDouble(GetAnalog(“4”, 5000));

}

else

{

// The temperature will need to be converted from Celcius to Fahrenheit.

// Communication between the system and the machine is slow, give it 5000

// milliseconds, 5 seconds, to return with a response.

return (Convert.ToDouble(GetAnalog(“4”, 5000)) \* 1.8 + 32);

}

}

#### string [] TigerStopAPI.TigerStop\_IO.GetCounter ()

Returns the 25 counter values tracked by the amp.

##### Returns:

A 'string' array of 25 entries that hold each of the 25 counter values tracked by the amp.

#### Example:

public void DisplayCounter()

{

// Get the array of strings.

string[] printStrings = io.GetCounter();

// Display each of the strings.

foreach (string s in printStrings)

{

Console.WriteLine(s);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetCounter (int *subCommand*)

Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | An 'int' that represents the desired subcommand value to retrieve. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in an expected time frame.

#### Example:

// A function to get the up time of the amplifier.

public double GetAmpTemp(bool inMinutes)

{

if (inMinutes)

{

// The machine returns the uptime in milliseconds, some conversion is needed to get

// minutes.

return (Convert.ToDouble(GetCounter(1)) / 1000 / 60)

}

// Just return it in seconds.

else

{

return (Convert.ToDouble(GetCounter(1)) / 1000);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetCounter (int *subCommand*, int *timeout*)

Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | An 'int' that represents the desired subcommand value to retrieve. |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in the expected time frame of 'timeout'.

#### Example:

// A function to get the up time of the amplifier.

public double GetAmpTemp(bool inMinutes)

{

if (inMinutes)

{

// The machine returns the uptime in milliseconds, some conversion is needed to get

// minutes. Communication is slow between the system and the machine, give it 5000

// milliseconds, 5 seconds, to respond.

return (Convert.ToDouble(GetCounter(1, 5000)) / 1000 / 60)

}

// Just return it in seconds.

else

{

return (Convert.ToDouble(GetCounter(1, 5000)) / 1000);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetCounter (string *subCommand*)

Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | A 'string' that represents the desired subcommand value to retrieve. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in an expected time frame.

#### Example:

// A function to get the up time of the amplifier.

public double GetAmpTemp(bool inMinutes)

{

if (inMinutes)

{

// The machine returns the uptime in milliseconds, some conversion is needed to get

// minutes.

return (Convert.ToDouble(GetCounter(“1”)) / 1000 / 60)

}

// Just return it in seconds.

else

{

return (Convert.ToDouble(GetCounter(“1”)) / 1000);

}

}

#### string TigerStopAPI.TigerStop\_IO.GetCounter (string *subCommand*, int *timeout*)

Returns the counter value related to the counter subcommand, 1-25, 'subCommand' matches. Waiting for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *subCommand* | A 'string' that represents the desired subcommand value to retrieve. |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'string' that represents the returned value of the subcommand at 'subCommand' or 'null' if a response is not received in the expected time frame of 'timeout'.

#### Example:

// A function to get the up time of the amplifier.

public double GetAmpTemp(bool inMinutes)

{

if (inMinutes)

{

// The machine returns the uptime in milliseconds, some conversion is needed to get

// minutes. Communication is slow between the system and the machine, give it 5000

// milliseconds, 5 seconds, to respond.

return (Convert.ToDouble(GetCounter(“1”, 5000)) / 1000 / 60)

}

// Just return it in seconds.

else

{

return (Convert.ToDouble(GetCounter(“1”, 5000)) / 1000);

}

}

#### string [] TigerStopAPI.TigerStop\_IO.GetLog ()

Returns the 20 most recent command and error log entries.

##### Returns:

A 'string' array of 20 entries that hold each of the 20 command and error log entries.

#### Example:

public void DisplayLog()

{

// Get the array of strings.

string[] printStrings = io.GetLog();

// Display each of the strings.

foreach (string s in printStrings)

{

Console.WriteLine(s);

}

}

#### string [] TigerStopAPI.TigerStop\_IO.GetLog (int *logIndex*)

Returns the 20 most recent command and error log entries starting at 'logIndex' and going back.

##### Parameters:

|  |  |
| --- | --- |
| *logIndex* | An 'int' that designates the log index to start from. |

##### Returns:

A 'string' array of 20 entries that hold each of the command and error log entries starting at 'logIndex'.

#### Example:

public void DisplayLog()

{

// Get the array of strings for log entries 20 - 40.

string[] printStrings = io.GetLog(20);

// Display each of the strings.

foreach (string s in printStrings)

{

Console.WriteLine(s);

}

}

#### string [] TigerStopAPI.TigerStop\_IO.GetLog (string *logIndex*)

Returns the 20 most recent command and error log entries starting at 'logIndex' and going back.

##### Parameters:

|  |  |
| --- | --- |
| *logIndex* | An 'string' that designates the log index to start from. |

##### Returns:

A 'string' array of 20 entries that hold each of the command and error log entries starting at 'logIndex'.

#### Example:

public void DisplayLog()

{

// Get the array of strings for log entries 20 - 40.

string[] printStrings = io.GetLog(“20”);

// Display each of the strings.

foreach (string s in printStrings)

{

Console.WriteLine(s);

}

}

#### double TigerStopAPI.TigerStop\_IO.GetPosition ()

Returns the current position of the machine.

##### Returns:

A 'double' that represents the returned current position from the machine or 'NaN ' if no response is received in an expected time frame.

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// Move the machine out to allow material to be loaded.

io.MoveTo(100);

// Get the postion of the machine to double check it’s in the correct position.

double pos = GetPosition();

// The machine isn’t perfect, but if it’s within an acceptable distance, then it’s in position.

if (pos != double.NaN && (pos > 99.9 && pos < 100.1))

{

Console.WriteLine(“In Position”);

}

#### double TigerStopAPI.TigerStop\_IO.GetPosition (int *timeout*)

Returns the current position of the machine, waiting for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'double' that represents the returned current position from the machine or ‘NaN' if no response is received in the expected time frame of 'timeout'.

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// Move the machine out to allow material to be loaded.

io.MoveTo(100);

// Get the postion of the machine to double check it’s in the correct position. Communication is slow

// between the system and the machine, give it 5000 milliseconds, 5 seconds, to respond.

double pos = GetPosition(5000);

// The machine isn’t perfect, but if it’s within an acceptable distance, then it’s in position.

if (pos != double.NaN && (pos > 99.9 && pos < 100.1))

{

Console.WriteLine(“In Position”);

}

#### string TigerStopAPI.TigerStop\_IO.GetSetting (int *settingIndex*)

Returns the value of a desired setting at 'settingIndex'.

##### Parameters:

|  |  |
| --- | --- |
| *settingIndex* | An 'int' that relates to the index of the desired setting to retrieve. |

##### Returns:

A 'string' containing the value of the setting at 'settingIndex' or 'null' if a response wasn't received in an expected time frame.

#### Example:

public bool IsUsingMetric()

{

bool isMetric = false;

// GetSetting() returns a string, convert it and check what value was returned. A ‘1’ means setting

// ‘MMEnable’, at index 44, is true and the machine is in metric.

if (Convert.ToDouble(GetSetting(44)) == 1)

{

isMetric = true;

}

// A ‘0’ means setting ‘MMEnable’, at index 44, is false and the machine is in imperial.

else if (Convert.ToDouble(GetSetting(44)) == 0)

{

isMetric = false;

}

return isMetric;

}

#### string TigerStopAPI.TigerStop\_IO.GetSetting (string *settingName*)

Returns the value of the desired setting whose name matches 'settingName'.

##### Parameters:

|  |  |
| --- | --- |
| *settingName* | A 'string' that matches the name of the desired setting to retrieve. |

##### Returns:

A 'string' containing the value of the setting whose name matches 'settingName' or 'null' if a response wasn't received in an expected time frame.

#### Example:

public bool IsUsingMetric()

{

bool isMetric = false;

// GetSetting() returns a string, convert it and check what value was returned. A ‘1’ means setting

// ‘MMEnable’ is true and the machine is in metric.

if (Convert.ToDouble(GetSetting(“MMEnable”)) == 1)

{

isMetric = true;

}

// A ‘0’ means setting ‘MMEnable’ is false and the machine is in imperial.

else if (Convert.ToDouble(GetSetting(“MMEnable”)) == 0)

{

isMetric = false;

}

return isMetric;

}

#### string TigerStopAPI.TigerStop\_IO.GetSetting (int *settingIndex*, int *timeout*)

Returns the value of the desired setting at 'settingIndex', waiting for 'timeout's duration for a response.

##### Parameters:

|  |  |
| --- | --- |
| *settingIndex* | An 'int' that relates to the index of the desired setting to retrieve. |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'string' containing the value of the setting at 'settingIndex' or 'null' if a response was not received in the expected time frame of 'timeout'.

#### Example:

public bool IsUsingMetric()

{

bool isMetric = false;

// GetSetting() returns a string, convert it and check what value was returned. A ‘1’ means setting

// ‘MMEnable’, at index 44, is true and the machine is in metric. Communication is slow between

// the system and the machine, give it 5000 milliseconds, 5 seconds, to respond.

if (Convert.ToDouble(GetSetting(44, 5000)) == 1)

{

isMetric = true;

}

// A ‘0’ means setting ‘MMEnable’, at index 44, is false and the machine is in imperial.

else if (Convert.ToDouble(GetSetting(44, 5000)) == 0)

{

isMetric = false;

}

return isMetric;

}

#### string TigerStopAPI.TigerStop\_IO.GetSetting (string *settingName*, int *timeout*)

Returns the value of the setting whose name matches 'settingName', waiting for 'timeout's duration for a response.

##### Parameters:

|  |  |
| --- | --- |
| *settingName* | A 'string' that matches the name of the desired setting to retrieve. |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

A 'string' containing the value of the setting whose name matches 'settingName' or 'null' if a response wasn't received in the expected time frame of 'timeout'.

#### Example:

public bool IsUsingMetric()

{

bool isMetric = false;

// GetSetting() returns a string, convert it and check what value was returned. A ‘1’ means setting

// ‘MMEnable’ is true and the machine is in metric. Communication is slow between

// the system and the machine, give it 5000 milliseconds, 5 seconds, to respond.

if (Convert.ToDouble(GetSetting(“MMEnable”, 5000)) == 1)

{

isMetric = true;

}

// A ‘0’ means setting ‘MMEnable’ is false and the machine is in imperial.

else if (Convert.ToDouble(GetSetting(“MMEnable”, 5000)) == 0)

{

isMetric = false;

}

return isMetric;

}

#### int TigerStopAPI.TigerStop\_IO.GetStatus ()

Returns the current status of the machine.

##### Returns:

An 'int' that represents the returned status of the machine or '-1' if no response is received in an expected time frame.

#### Example:

public bool IsMachineHalted()

{

bool isHalted = false;

// A ‘0’ status signals the machine is halted.

if (GetStatus() == 0)

{

isHalted = true;

}

// Any other status means that the machine is in motion or the status was not able to be interpreted.

else

{

isHalted = false;

}

return isHalted;

}

#### int TigerStopAPI.TigerStop\_IO.GetStatus (int *timeout*)

Returns the current status of the machine, waiting for the duration of 'timeout' for a response.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the function will wait for a response. |

##### Returns:

An 'int' that represents the returned status of the machine or '-1' if no response is received in the expected time frame of 'timeout'.

#### Example:

public bool IsMachineHalted()

{

bool isHalted = false;

// A ‘0’ status signals the machine is halted. Communication between the system and the machine

// is slow, give it 5000 milliseconds, 5 seconds, to respond.

if (GetStatus(5000) == 0)

{

isHalted = true;

}

// Any other status means that the machine is in motion or the status was not able to be interpreted.

else

{

isHalted = false;

}

return isHalted;

}

#### void TigerStopAPI.TigerStop\_IO.HomeDevice ()

Runs the home routine to return the machine to the home position, and enable the drive if it is disabled.

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// Now that the system is connected, home the machine so that it is ready to use.

io.HomeDevice();

// Homing the machine takes a while, check back on its status every couple of seconds.

while (GetStatus() != 0)

{

Thread.Sleep(2000);

}

// Move the machine out to allow material to be loaded.

io.MoveTo(100);

#### bool TigerStopAPI.TigerStop\_IO.IO\_Connection (int *ioNum*, bool *onOff*)

Turns the first IO connection on the IO panel on or off depending on ‘onOff’.

##### Parameters:

|  |  |
| --- | --- |
| *ioNum* | An ‘int’ denoting which IO connection, 1 – 12, on the IO panel to change. If ‘ioNum’ is outside the range 1 – 12, ‘false’ is returned. |
| *onOff* | A ‘bool’ that signals whether to turn the connection on if ‘onOff’ is ‘true’ and off if ‘onOff’ is ‘false’. |

##### Returns:

A 'bool' denoting whether an acknowledgment was received for the command.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to 100 inches to load material.

io.MoveTo(100);

// Turn the load signal IO on to let an auto-loader know to begin loading material.

io.IO\_Connection(12, true);

#### bool TigerStopAPI.TigerStop\_IO.IO\_Connection (int *ioNum*, bool *onOff*, int *timeout*)

Turns the first IO connection on the IO panel on or off depending on ‘onOff’, waiting for the duration of ‘timeout’ for a response.

##### Parameters:

|  |  |
| --- | --- |
| *ioNum* | An ‘int’ denoting which IO connection, 1 – 12, on the IO panel to change. If ‘ioNum’ is outside the range 1 – 12, ‘false’ is returned. |
| *onOff* | A ‘bool’ that signals whether to turn the connection on if ‘onOff’ is ‘true’ and off if ‘onOff’ is ‘false’. |
| *timeout* | An 'int' representing the desired timeout value in milliseconds the event will wait for a response. |

##### Returns:

A 'bool' denoting whether an acknowledgment was received for the command.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to 100 inches to load material.

io.MoveTo(100);

// Turn the load signal IO on to let an auto-loader know to begin loading material. Communication is slow

// between the system and the machine give it 2000 milliseconds, 2 seconds, to respond.

io.IO\_Connection(12, true, 2000);

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (double *position*)

Sends a move command to the machine to move to the desired position waiting for the move to finish.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A 'double' denoting the desired position to move to. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut.

io.MoveTo(100 – i);

// Cut the material.

io.CycleTool();

}

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (double *position*, int *timeout*)

Sends a move command to the machine to move to the desired position, waiting for the move to finish or for the given timeout duration before returning.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A 'double' denoting the desired position to move to. |
| *timeout* | An 'int' denotes the number of milliseconds to timeout on. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut. The movement needs to be completed in 10000

// milliseconds, 10 seconds, otherwise the process needs to stop.

io.MoveTo(100, 10000);

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut. The movement needs to be completed in 10000

// milliseconds, 10 seconds, otherwise the process needs to stop.

if (io.MoveTo(100 – i, 10000))

{

// Cut the material.

io.CycleTool();

}

else

{

break;

}

}

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (double *position*, ref BackgroundWorker *b*)

Sends a move command to the machine to move to the desired position.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A 'double' denoting the desired position to move to. |
| *b* | A ‘BackgroundWoker’ that is running MoveTo() that may signal an impending cancellation. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut. Pass the background worker down so

// that MoveTo() can cancel if there’s a cancellation pending.

io.MoveTo(100 – i, bkgndMakeCuts);

CycleTool();

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (double *position*, int *timeout*, ref BackgroundWorker *b*)

Sends a move command to the machine to move to the desired position.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A 'double' denoting the desired position to move to. |
| *timeout* | An ‘int’ that denotes the number of milliseconds to timeout on. |
| *b* | A ‘BackgroundWoker’ that is running MoveTo() that may signal an impending cancellation. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut. The movement needs to occur within

// 10000 milliseconds, 10 seconds, otherwise the process needs to cancel. Pass the

// background worker down so that MoveTo() can cancel if there’s a cancellation

// pending.

if (!io.MoveTo(100 – i, 10000, bkgndMakeCuts))

{

break;

}

CycleTool();

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (string *position*)

Sends a move command to the machine to move to the desired position waiting for the move to finish.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A 'string' denoting the desired position to move to. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to 100 inches to load material.

io.MoveTo(“100”);

// Push the material in 10 inches to make it ready to cut.

io.MoveTo(“90”);

io.CycleTool();

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (string *position*, int *timeout*)

Sends a move command to the machine to move to the desired position, waiting for the move to finish or for the given timeout duration before returning.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A 'string' denoting the desired position to move to. |
| *timeout* | An 'int' that denotes the number of milliseconds to timeout on. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to 100 inches to load material.

io.MoveTo(“100”);

// Push the material in 10 inches to make it ready to cut. The movement needs to be completed in 10000

// milliseconds, 10 seconds, otherwise actions need to stop.

if (io.MoveTo(“90”, 10000))

{

io.CycleTool();

}

else

{

io.Stop();

}

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (string *position*, ref BackgroundWorker *b*)

Sends a move command to the machine to move to the desired position.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A ‘string’ denoting the desired position to move to. |
| *b* | A ‘BackgroundWoker’ that is running MoveTo() that may signal an impending cancellation. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut. Pass the background worker down so

// that MoveTo() can cancel if there’s a cancellation pending.

io.MoveTo(Convert.ToInt32(“100”) – i, bkgndMakeCuts);

CycleTool();

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### bool TigerStopAPI.TigerStop\_IO.MoveTo (string *position*, int *timeout*, ref BackgroundWorker *b*)

Sends a move command to the machine to move to the desired position.

##### Parameters:

|  |  |
| --- | --- |
| *position* | A 'string' denoting the desired position to move to. |
| *timeout* | An ‘int’ that denotes the number of milliseconds to timeout on. |
| *b* | A ‘BackgroundWoker’ that is running MoveTo() that may signal an impending cancellation. |

##### Returns:

A 'bool' that signals whether the move command was successfully completed.

#### Example:

#### Example:

private void BackgroundWorker bkgndMakeCuts(object sender, DoWorkEventArgs e)

{

// Simple cut list like loop that moves in 10 inches and cuts a piece.

for ( int i = 10; i < 100; i += 10)

{

// Push the material in 10 inches for another cut. The movement needs to occur within

// 10000 milliseconds, 10 seconds, otherwise the process needs to cancel. Pass the

// background worker down so that MoveTo() can cancel if there’s a cancellation

// pending.

if (!io.MoveTo(Convert.ToInt32(“100”) – i, 10000, bkgndMakeCuts))

{

break;

}

CycleTool();

}

}

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to load material to cut.

io.MoveTo(100);

// Run a background worker to allow the UI to keep running.

bkgndMakeCuts.RunWorkerAsync();

#### void TigerStopAPI.TigerStop\_IO.RandomLengthMeasure ()

Sends a measure command to the machine and waits until the material length has been determined.

NOTE: 'length' values will always be in imperial inches. If metric values are desired, uncomment conversion and modify as needed.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move to 100 inches to load material. The measure command will move in towards an edge sensor.

io.MoveTo(100);

double metric;

double length = io.RandomLengthMeasure()

if (length > -1)

{

// Because RandomLengthMeasure only returns imperial inches, convert ‘length’ to metric.

metric = length \* 25.4;

}

#### void TigerStopAPI.TigerStop\_IO.RandomLengthMeasure (int timeout)

Sends a measure command to the machine and waits until the material length has been determined.

NOTE: 'length' values will always be in imperial inches. If metric values are desired, uncomment conversion and modify as needed.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An ‘int’ representing the desired timeout value in milliseconds the event will wait for a response |

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move to 100 inches to load material. The measure command will move in towards an edge sensor.

io.MoveTo(100);

double metric;

// Specify the measure command must complete in 10000 milliseconds, 10 seconds, to continue operation.

double length = io.RandomLengthMeasure(10000)

if (length > -1)

{

// Because RandomLengthMeasure only returns imperial inches, convert ‘length’ to metric.

metric = length \* 25.4;

}

else

{

Console.WriteLine(“Command timed out”);

}

#### void TigerStopAPI.TigerStop\_IO.ScanDefectedLength ()

Sends a scan command to the machine and waits until all of the UV detected marks and the material length have been determined and returns them as a 'List'.

NOTE: 'marks' values are converted to imperial or metric automatically based on the current 'MMEnable' value.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move to 2 inches to load material. The scan command will move away, over the material.

io.MoveTo(2);

List<double> marks = io.ScanDefectedLength()

if (marks.Count > 0)

{

// The final mark, at index n – 1, is the total length of the material.

Console.WriteLine(“The material length is: “ + marks[marks.Count – 1]);

// All other values are positions in which a mark was seen.

Console.WriteLine(“There were “ + (marks.Count – 1) + “ marks on the material.”);

}

else

{

Console.WriteLine(“Command timed out”);

}

#### void TigerStopAPI.TigerStop\_IO.ScanDefectedLength (int timeout)

Sends a scan command to the machine and waits until all of the UV detected marks and the material length have been determined, or the duration of 'timeout', and returns them as a 'List'.

NOTE: 'marks' values are converted to imperial or metric automatically based on the current 'MMEnable' value.

##### Parameters:

|  |  |
| --- | --- |
| *timeout* | An ‘int’ representing the desired timeout value in milliseconds the event will wait for a response |

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move to 2 inches to load material. The scan command will move away, over the material.

io.MoveTo(2);

// Specify the scan command must complete in 10000 milliseconds, 10 seconds, to continue operation.

List<double> marks = io.ScanDefectedLength(10000)

if (marks.Count > 0)

{

// The final mark, at index n – 1, is the total length of the material.

Console.WriteLine(“The material length is: “ + marks[marks.Count – 1]);

// All other values are positions in which a mark was seen.

Console.WriteLine(“There were “ + (marks.Count – 1) + “ marks on the material.”);

}

else

{

Console.WriteLine(“Command timed out”);

}

#### void TigerStopAPI.TigerStop\_IO.Stop ()

Sends a stop command to the machine, ending any action its currently in the middle of.

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Move the machine out to 100 inches to load material.

io.MoveTo(“100”);

// Push the material in 10 inches to make it ready to cut. The movement needs to be completed in 10000

// milliseconds, 10 seconds, otherwise actions need to stop.

if (io.MoveTo(“90”, 10000))

{

io.CycleTool();

}

else

{

io.Stop();

}

#### bool TigerStopAPI.TigerStop\_IO.UpdateSetting (string *settingName*, double *newValue*)

Changes the setting value of 'settingName' to 'newValue'.

##### Parameters:

|  |  |
| --- | --- |
| *settingName* | A 'string' the represents the name of the setting to change. |
| *newValue* | A 'double' that represents the new value 'settingName' is being changed to. |

##### Returns:

A 'bool' that signals whether the setting was successfully changed.

#### Example:

public void ChangeUnits()

{

// Use GetSetting() to figure out what the current units are. If “MMEnable” is ‘1’, then call

// UpdateSetting() to change it to ‘0’.

if (Convert.ToDouble(GetSetting(44)) == 1)

{

UpdateSetting(“MMEnable”, 0);

}

// If “MMEnable” is ‘0’, then call UpdateSetting() to change it to ‘1’.

else

{

UpdateSetting(“MMEnable”, 1);

}

}

#### bool TigerStopAPI.TigerStop\_IO.UpdateSetting (string *settingName*, string *newValue*)

Changes the setting value of 'settingName' to 'newValue'.

##### Parameters:

|  |  |
| --- | --- |
| *settingName* | A 'string' the represents the name of the setting to change. |
| *newValue* | An 'string' that represents the new value 'settingName' is being changed to. |

##### Returns:

A 'bool' that signals whether the setting was successfully changed.

#### Example:

public void ChangeUnits()

{

// Use GetSetting() to figure out what the current units are. If “MMEnable” is ‘1’, then call

// UpdateSetting() to change it to ‘0’.

if (Convert.ToDouble(GetSetting(44)) == 1)

{

UpdateSetting(“MMEnable”, “0”);

}

// If “MMEnable” is ‘0’, then call UpdateSetting() to change it to ‘1’.

else

{

UpdateSetting(“MMEnable”, “1”);

}

}

#### bool TigerStopAPI.TigerStop\_IO.UpdateSetting (int *settingIndex*, double *newValue*)

Changes the setting value at 'settingIndex' to 'newValue'.

##### Parameters:

|  |  |
| --- | --- |
| *settingIndex* | An 'int' the represents the index of the setting to change. |
| *newValue* | An 'int' that represents the new value the setting is being changed to. |

##### Returns:

A 'bool' that signals whether the setting was successfully changed.

#### Example:

public void ChangeUnits()

{

// Use GetSetting() to figure out what the current units are. If “MMEnable”, at index 44, is ‘1’,

// then call UpdateSetting() to change it to ‘0’.

if (Convert.ToDouble(GetSetting(44)) == 1)

{

UpdateSetting(44, 0);

}

// If “MMEnable”, at index 44, is ‘0’, then call UpdateSetting() to change it to ‘1’.

else

{

UpdateSetting(44, 1);

}

}

#### bool TigerStopAPI.TigerStop\_IO.UpdateSetting (int *settingIndex*, string *newValue*)

Changes the setting value at 'settingIndex' to 'newValue'.

##### Parameters:

|  |  |
| --- | --- |
| *settingIndex* | An 'int' the represents the index of the setting to change. |
| *newValue* | A 'double' that represents the new value the setting is being changed to. |

##### Returns:

A 'bool' that signals whether the setting was successfully changed.

#### Example:

public void ChangeUnits()

{

// Use GetSetting() to figure out what the current units are. If “MMEnable”, at index 44, is ‘1’,

// then call UpdateSetting() to change it to ‘0’.

if (Convert.ToDouble(GetSetting(44)) == 1)

{

UpdateSetting(44, “0”);

}

// If “MMEnable”, at index 44, is ‘0’, then call UpdateSetting() to change it to ‘1’.

else

{

UpdateSetting(44, “1”);

}

}

#### void TigerStopAPI.TigerStop\_IO.WriteCommand (string *command*)

Sends 'command' to the machine to execute.

##### Parameters:

|  |  |
| --- | --- |
| *command* | A 'string' representing the command being sent to the machine to execute. |

#### Example:

TigerStop\_IO io = new TigerStop(9600, “COM1”);

// Send a command directly to the machine, without any use of flags or timeouts. Move the machine to 100

// inches.

WriteCommand(“mg100”);

// Cycle the tool.

WriteCommand(“mt”);

#### The documentation for this class was generated from the following file:

TigerStopAPI/TigerStop\_IO.cs

## TigerStopAPI.TigerStop\_Com Class Reference

Inherited by **TigerStopAPI.TigerStop\_IO**.

### Classes

* struct **LastAck**
* struct **LastCommand**

### Public Member Functions

* **TigerStop\_Com** (int baud, string comPort)

### Public Attributes

* string **serialNumber**
* bool **isRS232** = true
* bool **isLastConnected** = false
* string **comPortName**
* int **baudrate**

### Protected Member Functions

* void **QueueCommand** (string command)

*This function is the main interface between the rest of the system and the machine. Any commands that need to be sent to the machine runs through this command. It takes a 'string' command to send to the machine. If the system already has commands queued up, it will add the command to the queue, otherwise it will call* ***SendCommand()*** *to get the command processed immediately.*

* void **QueueCommand** (byte[] command)
* *This function is the main interface between the rest of the system and the machine. Any commands that need to be sent to the machine runs through this command. It takes a 'byte[]’ command to send to the machine. If the system already has commands queued up, it will add the command to the queue, otherwise it will call* ***SendCommand()*** *to get the command processed immediately.*
* void **ClearCommand** (bool allCommands)

*This function is used to clear out commands from the 'writeBuffer'. If 'allCommands' is 'true', it will clear all commands from the 'writeBuffer'.* ***Otherwise, it will only clear the first command from 'writeBuffer'.***

* void **GetSettings** ()

*This function is used to ask the machine for all of its settings and puts them into a list for future use.*

* void **ClearPort** ()

*This function is used to clear out the serial port, reading anything currently in the serial port.*

* void **ClosePort** ()

*Used to close the port when it is no longer in use.*

* void **OpenPort** ()

*This function takes the stored com port name and baud rate and attempts to open a serial connection to the desired com port.*

* bool **DetectTigerStop** ()

*This function sends the serial command query to the machine to get a hold of its serial number. If the serial number is valid, then* ***SerialPort\_DataReceived()*** *will signal the 'serialAck' to allow the function through and to return true.*

* bool **CheckConnection** ()

*This function goes through all of the necessary checks that ensures the system is connected to a machine. If all of the checks pass a 'bool' is returned 'true' denoting that the system has successfully connected to the machine.*

* void **ChangeSetting** (string command, int index)

*Takes a setting command and setting index to update the desired setting in 'settings' at 'index'.*

* void **ChangeSetting** (string command)

*Sends a setting change command to the machine.*

* void **InitializeTimeouts** ()

*Takes the currently saved timeout settings and initializes the timeouts to more expected timeouts.*

* void **LoadLight** (bool on)

*This function is used to write to the serial port to have the machine turn on the load signal light according to the 'bool' input.*

### Static Protected Member Functions

* static List< KeyValuePair< string, int > > **FindConnections** ()

*Opens each of the available comports at a number of baudrates and checks each for a potential connection to a TigerStop amp.*

### Properties

* bool **IsOpen** [get]
* bool **IsConnected** [get]
* DateTime **LastAckTime** [get]
* SerialPort **Port** [get, private set]
* double **Position** [get, private set]
* List< double > **Settings** [get]
* TimeSpan **TimeOut** [get, private set]

### Events

* EventHandler **SendData**
* EventHandler **AddSetting**
* PropertyChangedEventHandler **PropertyChanged**
* EventHandler **UpdateSetting**
* EventHandler **StopOperation**

### Private Member Functions

* void **SerialPort\_AddSetting** (object sender, EventArgs e)

*This event handler is used specifically with the SerialPort\_DataRecieved() event handler when the system is still in setup and obtaining all of the settings.*

* void **SerialPort\_UpdateSetting** (object sender, EventArgs e)

*This event handler is used specifically with the* ***SerialPort\_DataReceived()*** *event handler when the system is updating a specific setting. Upon retrieving the specific setting at 'settingIndex', if the returned value can be deciphered, it’s the new setting value, otherwise just keep the old value.*

* void **SerialPort\_DataReceived** (object sender, SerialDataReceivedEventArgs e)

*This is the main event handler, everything from the machine will be funneled through this event handler. Anytime the serial port buffer receives data, this the SerialPort.DataReceived event will fire and this event handler will be called to take in the data. This event handler is given its own thread to handle the data.*

* void **NotifyPropertyChanged** (string property)

*Basic property changed event handler.*

* void **HandleData** (string data)

*Takes in a string of data from* ***SerialPort\_DataReceived()*** *and parses it with any data in 'readBuffer' to determine if the machine has sent back an ack for us to decipher at any point.*

* void **HandleAck** ()

*Once* ***HandleData()*** *collates the data taken in from* ***SerialPort\_DataReceived()*** *and checks it for appropriate acks based on what the system is doing at the moment.*

* byte [] **CommandConverter** (string input)

*Used specifically to convert 'string's into hex byte commands to send to the machine.*

* void **SendCommand** ()

*This function takes the first command from 'writeBuffer' and, depending on the command, sends it to the machine through the proper functions.*

* void **MoveCommand** (byte[] moveCommand)

*Called if the first command seen by* ***SendCommand()*** *is a move command, determine what kind of move command is being sent and set the appropriate flags and timeouts before sending the command to the machine.*

* void **WriteToSerial** (byte[] command)

*This function writes the byte[] command to the machine over the serial port. Also tracks the last command that was sent, in case we need send it again.*

* void **WriteToSerialClean** (byte[] command)

*This function writes the byte[] command to the machine over the serial port. It does not track the last command.*

* void **RetryCommand** ()

*This function sends the last command to the machine in the case the machine did not register or complete the last command.*

* bool **CheckSamePosition** ()

*This function is used to determine if the position that the machine is at is the same as the last position that was queried. If the position is the same, the function returns 'true' if the position in the last acknowledgment is the same as the current position we know of. Otherwise, the function returns false.*

* void **CheckMovement** ()

*This function is used while the machine is moving to double check that the machine is, in fact, moving like it was told to.*

* void **ChangeFlags** (bool change)

*Changes all of the 'bool' flags to the value of 'change'.*

### Private Attributes

* AutoResetEvent **serialAck** = new AutoResetEvent(false)
* AutoResetEvent **updateAck** = new AutoResetEvent(false)
* BackgroundWorker **bkgndCycle** = new BackgroundWorker()
* List< string > **readBuffer** = new List<string>()
* List< byte[]> **writeBuffer** = new List<byte[]>()
* const double **HALTED** = 0
* const double **ACCEL** = 1
* const double **CONST\_VEL** = 2
* const double **DECEL** = 3
* const double **DRIVE\_DISABLED** = 4
* const double **LASH** = 5
* const double **WAIT\_TO\_MOVE** = 6
* const double **EMERGENCY\_STOP** = 7
* const double **SLEEP** = 8
* const double **MANUAL** = 9
* int **settingIndex**
* bool **isConnected** = false
* bool **isSetup** = true
* bool **isGettingSettings**
* bool **isDetectingTS** = false
* bool **isUpdatingSetting** = false
* bool **isMoving** = false
* bool **isMoveStart** = false
* bool **isCyclingTool** = false
* bool **isCycleStart** = false
* bool **isDmOff** = false
* bool **isDmOn** = false
* bool **isScanning** = false
* bool **isMeasuring** = false
* List< DateTime > **ackTimes** = new List<DateTime>()
* List< TimeSpan > **mtAckTimes** = new List<TimeSpan>()
* List< double > **settings** = new List<double>()
* List< double > **scanMarks** = new List<double>()
* SerialPort **port** = new SerialPort()
* TimeSpan **timeout**
* TimeSpan **mtTimeout** = TimeSpan.FromSeconds(10)
* TimeSpan **scanTimeout** = TimeSpan.FromSeconds(15)
* TimeSpan **homeTimeout** = TimeSpan.FromSeconds(60)

### Static Private Attributes

* static readonly byte [] **moveToolCommand** = { 0x6d, 0x74, 0x0d, 0x0a }
* static readonly byte [] **moveHomeCommand** = { 0x6d, 0x68, 0x0d, 0x0a }
* static readonly byte [] **moveStopCommand** = { 0x6d, 0x73, 0x0d, 0x0a }
* static readonly byte [] **moveEStopCommand** = { 0x6d, 0x65, 0x0d, 0x0a }
* static readonly byte [] **positionQueryCommand** = { 0x70, 0x0d, 0x0a }
* static readonly byte [] **scanCommand** = { 0x02, 0x52, 0x00, 0x01, 0x80, 0x00, 0x00, 0x00, 0x0d, 0x0a }
* static readonly byte [] **statusQueryCommand** = { 0x73, 0x0d, 0x0a }
* static readonly byte [] **serialQueryCommand** = { 0x04, 0x31, 0x0d, 0x0a }
* static byte [] **loadSignalOn** = { 0x77, 0x77, 0x20, 0x30, 0x78, 0x38, 0x30, 0x30, 0x20, 0x30, 0x78, 0x38, 0x30, 0x30, 0x0a, 0x0d }
* static byte [] **loadSignalOff** = { 0x77, 0x77, 0x20, 0x30, 0x20, 0x30, 0x78, 0x38, 0x30, 0x30, 0x0a, 0x0d }
* static **LastCommand** **lastCommand** = new **LastCommand**(null, DateTime.Now)
* static **LastAck** **lastAck** = new **LastAck**(null, DateTime.Now)
* static double **position**
* static double **targetPosition**

### Detailed Description

Provides serial communication ability focused on a connection first communication infrastructure. Creates a FIFO command queue with each command executed in order they are received. The system will wait for an acknowledgement after each command sent before executing the next command in the queue. This class is the buffer between the rest of the system and communications with the machine, taking all of the raw data in and sending it up to **TigerStop\_IO** in a more understandable fashion.

### Member Function Documentation

This function list is sorted roughly, if not completely, alphabetically:

#### void TigerStopAPI.TigerStop\_Com.ChangeFlags (bool *change*)[private]

Changes all of the 'bool' flags to the value of 'change'.

##### Parameters:

|  |  |
| --- | --- |
| *change* | A 'bool' representing the value to change all of the flags to. |

#### Example:

…

if (cmd.SequenceEqual(moveStopCommand))

{

// Send the stop command to the machine.

WriteToSerial(moveStopCommand);

// The machine is stopping, any queued commands need to be cleared to prevent further actions.

ClearCommand(true);

// The machine is stopping, it’s no longer performing any actions and any flags that were ‘true’

// need to be changed to ‘false’ to reflect the stopped state.

ChangeFlags(false);

}

…

#### void TigerStopAPI.TigerStop\_Com.ChangeSetting (string *command*, int *index*)[protected]

Takes a setting command and setting index to update the desired setting in 'settings' at 'index'.

##### Parameters:

|  |  |
| --- | --- |
| *command* | A 'string' that will be sent to the machine to change the setting in the command to the desired value. |
| *index* | An 'int' that denotes where in 'settings' the new setting value will be saved. |

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// Change the ‘MMEnable’ setting to 1, setting the machine to convert units to millimeters.

// Send the command ‘d44 1’, that will be converted for the amp, and the setting index ’44’,

// which will tell ‘io’ which setting to change if the amp replies.

io.ChangeSetting(“d44 1”, 44);

#### void TigerStopAPI.TigerStop\_Com.ChangeSetting (string *command*)[protected]

Sends a setting change command to the machine.

##### Parameters:

|  |  |
| --- | --- |
| *command* | A 'string' that will be sent to the machine to change the setting in the command to the desired value. |

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// Change the ‘MMEnable’ setting to 1, setting the machine to convert units to millimeters.

// Send the command ‘d44 1’ to the amp. Private int ‘settingIndex’ is set to 0 in the function,

// so ‘io’ does not have any internal settings to update.

io.ChangeSetting(“d44 1”);

#### bool TigerStopAPI.TigerStop\_Com.CheckConnection ()[protected]

This function goes through all of the necessary checks that ensures the system is connected to a machine. If all of the checks pass a 'bool' is returned 'true' denoting that the system has successfully connected to the machine.

##### Returns:

Returns a bool denoting whether we were able to connect to a machine with a valid enable code.

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// CheckConnection will make sure that the connection is open and that the amp is setup to receive serial

// commands from an outside source.

if (io.CheckConnection())

{

Console.WriteLine(“Connected.”);

}

else

{

Console.WriteLine(“Not Connected.”);

}

#### void TigerStopAPI.TigerStop\_Com.CheckMovement ()[private]

This function is used while the machine is moving to double check that the machine is, in fact, moving like it was told to.

#### Example:

…

// If the position hasn’t changed while ‘isMoving’ is ‘true’, check that the machine is moving.

if (!CheckSamePosition())

{

// CheckMovement() will query the status of the machine to determine if it is still in motion or if is

// at the target position.

CheckMovement();

}

…

#### bool TigerStopAPI.TigerStop\_Com.CheckSamePosition ()[private]

This function is used to determine if the position that the machine is at is the same as the last position that was queried. The function returns 'true' if the position in the last acknowledgment is the same as the current position we know of. Otherwise, the function returns false.

##### Returns:

Returns a bool denoting whether the machine is in the same place as the last time position was queried.

#### Example:

…

// The machine is in the middle of a movement.

else if (isMoving || isCyclingTool)

{

lastAck.Acknowledgment = string.Join(“”, radeBuffer.ToArray());

lastAck.TimeReceived = DateTime.Now;

// While it’s expected to be moving, check that the position is changing to guarantee that the

// machine is, in fact, moving.

if (!CheckSamePosition())

{

CheckMovement();

}

…

#### void TigerStopAPI.TigerStop\_Com.ClearCommand (bool *allCommands*)[protected]

This function is used to clear out commands from the 'writeBuffer'. If 'allCommands' is 'true', it will clear all commands from the 'writeBuffer'. **Otherwise, it will only clear the first command from 'writeBuffer'.**

##### Parameters:

|  |  |
| --- | --- |
| *allCommands* | A 'bool' that determines whether to clear the first command in the list or to clear all of the commands from the list. |

#### Example:

…

if (cmd.SequenceEqual(moveStopCommand))

{

// Send the stop command to the machine.

WriteToSerial(moveStopCommand);

// The machine is stopping, any queued commands need to be cleared to prevent further actions.

ClearCommand(true);

// The machine is stopping, it’s no longer performing any actions and any flags that were ‘true’

// need to be changed to ‘false’ to reflect the stopped state.

ChangeFlags(false);

}

…

#### void TigerStopAPI.TigerStop\_Com.ClearPort ()[protected]

This function is used to clear out the serial port, reading anything currently in the serial port.

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// Serial\_DataRecieved() uses Serial.ReadLine() to retrieve data from the serial port. Meaning, if any

// command is not ‘\n’ terminated it will remain in the port. Early versions of the ‘cs’ command were not

// ‘\n’ delimited and got stuck in the port.

io.QueueCommand(“cs”);

// ClearPort() uses Serial.ReadExisting() to read out any and all data out of the port, clearing it.

io.ClearPort();

#### void TigerStopAPI.TigerStop\_Com.ClosePort ()[protected]

Used to close the port when it is no longer in use.

#### Example:

…

// If the connection is open to a machine, close it down so that a connection can be opened to another one.

if (io.CheckConnection())

{

io.ClosePort();

}

…

#### byte [] TigerStopAPI.TigerStop\_Com.CommandConverter (string *input*)[private]

Used specifically to convert 'string's into hex byte commands to send to the machine.

##### Parameters:

|  |  |
| --- | --- |
| *input* | A 'string' that will be translated. |

##### Returns:

Returns a 'byte[]' to be used as a hex byte command by the machine.

#### Example:

Public void QueueCommand(string command)

{

…

else if(writeBuffer.Count != 0)

{

writeBuffer.Add(CommandConverter(command));

}

else

{

// The machine only understands byte commands, convert ‘string’ command into ‘byte’s

// before sending it to the machine.

writeBuffer.Add(CommandConverter(command));

SendCommand();

}

}

#### bool TigerStopAPI.TigerStop\_Com.DetectTigerStop ()[protected]

This function sends the serial command query to the machine to get a hold of its serial number. If the serial number is valid, then **SerialPort\_DataReceived()** will signal the 'serialAck' to allow the function through and to return true.

##### Returns:

Returns a ‘bool’ that signals whether or not the system detect a TigerStop machine on the other end of the connection.

#### Example:

TigerStop\_IO io = new TigerStop\_IO(9600, “COM1”);

// DetectTigerStop will send a serial query across the connection to retrieve a valid serial number from // the machine.

if (io.DetectTigerStop())

{

Console.WriteLine(“TigerStop Machine Detected.”);

}

else

{

Console.WriteLine(“No TigerStop Machine Detected.”);

}

#### static List<KeyValuePair<string, int> > TigerStopAPI.TigerStop\_Com.FindConnections ()[static], [protected]

Opens each of the available comports at a number of baudrates and checks each for a potential connection to a TigerStop amp.

##### Returns:

A 'List' of 'KeyValuePair's with comport names as 'string' keys and baudrate 'int' values of potential connections.

#### Example:

TigerStop\_IO.cs:

// Make a public wrapper function around FindConnections() that the user can access.

public static List<KeyValuePair<string, int>> Connections()

{

return FindConnections();

}

Program.cs

List<KeyValuePair<string, int>> con = new List<KeyValuePair<string, int>>();

// Make a list of available connections.

con = TigerStop\_IO.Connections();

if(con.Count > 0)

{

// Print out all of the connections that have a machine on the other end that is able to communicate.

foreach (KeyValuePair<string, int> c in con)

{

Console.WriteLine(“Comport: “ + c.key + “ | Baudrate: “ + c.Value);

}

}

#### void TigerStopAPI.TigerStop\_Com.GetSettings ()[protected]

This function is used to ask the machine for all of its settings and puts them into a list for future use.

#### Example:

public void GetAllSettings()

{

// Pull settings into TigerStop\_Com’s ‘List<double> settings’.

base.GetSettings();

for (int i = 0; (i < base.Settings) && (i < settingNames.Length); i++)

{

// Add ‘TigerStop\_Com’s ‘settings’ into ‘TigerStop\_IO’s ‘settings’ to allow access to the

// user.

settings.Add(base.Settings[i]);

}

}

#### void TigerStopAPI.TigerStop\_Com.HandleAck ()[private]

Once **HandleData()** collates the data taken in from **SerialPort\_DataReceived()** and checks it for appropriate acks based on what the system is doing at the moment.

#### Example:

private void HandleData(string data)

{

if (!isScanning)

{

HandleAck();

}

…

}

#### void TigerStopAPI.TigerStop\_Com.HandleData (string *data*)[private]

Takes in a string of data from **SerialPort\_DataReceived()** and parses it with any data in 'readBuffer' to determine if the machine has sent back an ack for us to decipher at any point.

##### Parameters:

|  |  |
| --- | --- |
| *data* | A 'string' containing data from the serial port to send off to **HandleAck()** depending the systems current status. |

#### Example:

private void SerialPort\_DataReceived(object sender, SerialDataReceivedEventArgs e)

{

SerialPort sp = (SerialPort) sender;

try

{

string data = sp.ReadLine();

if (!isSetup)

{

// If the program is ready to take data in, Serial.ReadLine() reads in a ‘string’

// send the ‘string’ to HandleData();

HandleData(data);

}

…

}

…

}

#### void TigerStopAPI.TigerStop\_Com.InitializeTimeouts ()[protected]

Takes the currently saved timeout settings and initializes the timeouts to more expected timeouts.

#### Example:

// TigerStop\_Com constructor that opens the connections and initializes all of the settings.

TigerStop\_Com(int baud, string comPort)

{

// Take in the baudrate and comport name for OpenPort().

this.baudrate = baud;

this.comPortName = comPort;

port.DataReceived += SerialPort\_DataReceived;

AddSetting += SerialPort\_AddSetting;

UpdateSetting += SerialPort\_UpdateSetting;

// Set up the serial connection.

OpenPort();

// Pull in the machine’s settings.

GetSettings();

// Using the settings that were just pulled in, initialize the timeouts for the functions that wait on

// the machine.

InitializeTimeouts();

}

#### void TigerStopAPI.TigerStop\_Com.LoadLight (bool *on*)[protected]

This function is used to write to the serial port to have the machine turn on the load signal light according to the 'bool' input.

##### Parameters:

|  |  |
| --- | --- |
| *on* | A 'bool' that denotes whether to turn on or turn off the load signal light on the machine. |

#### Example:

TigerStop\_IO.cs:

public void EnableLoadLight(bool enable)

{

base.LoadLight(enable);

}

Program.cs:

…

// Move the machine out 100 inches to load material onto the machine to push.

io.MoveTo(100);

// Turn on the load light so that an auto loader can be signaled to load the material.

io.EnableLoadLight(true);

…

#### void TigerStopAPI.TigerStop\_Com.MoveCommand (byte [] *moveCommand*)[private]

Called if the first command seen by **SendCommand()** is a move command, determine what kind of move command is being sent and set the appropriate flags and timeouts before sending the command to the machine.

##### Parameters:

|  |  |
| --- | --- |
| *moveCommand* | A 'byte[]' command that will be used to determine which move command is being sent. |

#### Example:

…

// Take the first command to send out of the write buffer.

byte[] send = writeBuffer[0];

// If the first byte matches the byte value of ‘m’, it’s a move command.

if (send[0] == 0x06d)

{

// Let MoveCommand interpret the command, using the proper timeouts and flags.

MoveCommand(send);

}

…

#### void TigerStopAPI.TigerStop\_Com.NotifyPropertyChanged (string *property*)[private]

Basic property changed event handler.

##### Parameters:

|  |  |
| --- | --- |
| *property* | The 'string' name of the property that was changed, to be sent out for others to identify and decide what to do with it. |

#### Example:

public bool IsConnected

{

get

{

return isConnected;

}

// Let a subscriber to the PropertyChanged event be notified when ‘isConnected’ is changed. This

// way they can know if a connection has been made.

private set

{

isConnected = value;

NotifyPropertyChanged(“IsConnected”);

}

}

#### void TigerStopAPI.TigerStop\_Com.OpenPort ()[protected]

This function takes the stored com port name and baud rate and attempts to open a serial connection to the desired com port.

#### Example:

// TigerStop\_Com constructor that opens the connections and initializes all of the settings.

TigerStop\_Com(int baud, string comPort)

{

// Take in the baudrate and comport name for OpenPort().

this.baudrate = baud;

this.comPortName = comPort;

port.DataReceived += SerialPort\_DataReceived;

AddSetting += SerialPort\_AddSetting;

UpdateSetting += SerialPort\_UpdateSetting;

// Set up the serial connection using ‘baudrate’ and ‘comPortName’.

OpenPort();

// Pull in the machine’s settings.

GetSettings();

// Using the settings that were just pulled in, initialize the timeouts for the functions that wait on

// the machine.

InitializeTimeouts();

}

#### void TigerStopAPI.TigerStop\_Com.QueueCommand (string *command*)[protected]

This function is the main interface between the rest of the system and the machine. Any commands that need to be sent to the machine runs through this command. It takes a 'string' command to send to the machine. If the system already has commands queued up, it will add the command to the queue, otherwise it will call **SendCommand()** to get the command processed immediately.

##### Parameters:

|  |  |
| --- | --- |
| *command* | A 'string' that will be converted to a 'byte[]' command that will be sent to the machine. |

#### Example:

public bool MoveTo(string position)

{

bool isDone = false;

// Position is the argument for a move-go command, add ‘mg’ in front of it and queue it for

// sending to the machine.

base.QueueCommand(“mg” + position);

movingEvent.Reset();

movingEvent.WaitOne();

isDone = true;

return isDone;

}

#### void TigerStopAPI.TigerStop\_Com.QueueCommand (byte[] *command*)[protected]

This function is the main interface between the rest of the system and the machine. Any commands that need to be sent to the machine runs through this command. It takes a 'byte[]' command to send to the machine. If the system already has commands queued up, it will add the command to the queue, otherwise it will call **SendCommand()** to get the command processed immediately.

##### Parameters:

|  |  |
| --- | --- |
| *command* | A 'byte[]' command that will be sent to the machine. |

#### Example:

public List<double> ScanDefectedLength()

{

List<double> marks = new List<double>();

ackEvent.Reset();

// Because the scan command contains a number of bytes that are not easily represented as ASCII

// characters, send the command as a byte[] for simplicity.

base.QueueCommand(new byte[] { 0x02, 0x52, 0x00, 0x01, 0x80, 0x00, 0x00, 0x00, 0x0d, 0x0a });

ackEvent.WaitOne(-1);

try

{

marks = AckMessage.Split(‘;’).ToList().Select(s => double.Parse(s)).ToList();

}

catch

{

marks.Clear();

}

return marks;

}

#### void TigerStopAPI.TigerStop\_Com.RetryCommand ()[private]

This function sends ‘lastCommand.Command’ to the machine in the case the machine did not register or complete the last command. **Caution is required when using this function as ‘lastCommand.Command’ may cause the machine to act unexpectedly, such as cycling the tool or moving when the user is unprepared.**

#### Example:

…

// If SerialPort\_DataReceived() received a ‘NACK’ from the machine, then the command that was just sent

// wasn’t received properly by the machine and needs to be resent.

else if (string.Join(“”, readBuffer.ToArray()).Contains(“NACK”))

{

// Go to ‘lastCommad’ to see what was just sent to the machine and resend it.

RetryCommand();

}

…

#### void TigerStopAPI.TigerStop\_Com.SendCommand ()[private]

This function takes the first command from 'writeBuffer' and, depending on the command, sends it to the machine through the proper functions.

#### Example:

…

// A tool cycle finished ack was received. Change the tool cycle specific flags and update ‘lastAck’.

if (isCyclingTool)

{

isDmOff = false;

isDmOn = false;

lastAck.Acknowledgment = “MTF”;

lastAck.TimeReceived = DateTime.Now;

SendData(this, new MessageEvent(“MTF”));

isCyclingTool = false;

}

// Clear the ack out of the buffer.

readBuffer.Clear();

// The move tool command in ‘writeBuffer’ was just finished, have ClearCommand() remove it from the

// queue.

ClearCommand(false);

// Send the next command in the queue to the machine.

SendCommand();

…

#### void TigerStopAPI.TigerStop\_Com.SerialPort\_AddSetting (object *sender*, EventArgs *e*)[private]

This event handler is used specifically with the SerialPort\_DataRecieved() event handler when the system is still in setup and obtaining all of the settings.

##### Parameters:

|  |  |
| --- | --- |
| *sender* | The object that called SerialPort\_AddSetting() to handle an event. |
| *e* | The arguments sent by the sender for the event handler to use. |

#### void TigerStopAPI.TigerStop\_Com.SerialPort\_DataReceived (object *sender*, SerialDataReceivedEventArgs *e*)[private]

This is the main event handler, everything from the machine will be funneled through this event handler. Anytime the serial port buffer receives data, the SerialPort.DataReceived event will fire and this event handler will be called to take in the data. This event handler is given its own thread to handle the data. **The SerialPort.DataReceived event is handled by the underlying .NET Framework code and operating system and SerialPort\_DataReceived may not always be called when expected.**

##### Parameters:

|  |  |
| --- | --- |
| *sender* | The object that called SerialPort\_DataReceived() to handle an event. |
| *e* | The arguments sent by the sender for the event handler to use. |

#### void TigerStopAPI.TigerStop\_Com.SerialPort\_UpdateSetting (object *sender*, EventArgs *e*)[private]

This event handler is used specifically with the **SerialPort\_DataReceived()** event handler when the system is updating a specific setting. Upon retrieving the specific setting at 'settingIndex', if the returned value can be deciphered, its the new setting value, otherwise just keep the old value.

##### Parameters:

|  |  |
| --- | --- |
| *sender* | The object that called SerialPort\_UpdateSetting() to handle an event. |
| *e* | The arguments sent by the sender for the event handler to use. |

#### void TigerStopAPI.TigerStop\_Com.WriteToSerial (byte [] *command*)[private]

This function writes the byte[] command to the machine over the serial port. Also tracks the last command that was sent, in case we need send it again.

##### Parameters:

|  |  |
| --- | --- |
| *command* | A 'byte' array of converted 'char' characters that the machine will recognize as an actionable command. |

#### Example:

private void SendCommand()

{

…

// If a scan command is being sent, use the proper timeout and flags.

else if (send.SequenceEqual(scanCommand))

{

TimeOut = Timeout.InfiniteTimeSpan;

isScanning = true;

// Write the ‘byte[]’ command to the machine across the serial connection.

WriteToSerial(scanCommand);

}

// Otherwise, it’s a command that doesn’t require and preparation.

else

{

// Write the ‘byte[]’ command to the machine across the serial connection.

WriteToSerial(send);

}

}

#### void TigerStopAPI.TigerStop\_Com.WriteToSerialClean (byte [] *command*)[private]

This function writes the byte[] command to the machine over the serial port. It does not track the last command.

##### Parameters:

|  |  |
| --- | --- |
| *command* | A 'byte' array of converted 'char' characters that the machine will recognize as an actionable command. |

#### Example:

protected void CheckSerialNumber()

{

bool isValudSerial = false;

isCheckSerial = true;

// Checking serial is generally part of the setup, the serial query will be sent but ‘lastCommand’

// won’t be updated. This ensures that if this command was sent while another command was

// waiting, an ack will be recognized for it rather than this new command, and RetryCommand()

// will know to call the more important command.

WriteToSerialClean(serialQueryCommand);

…

}

#### The documentation for this class was generated from the following file:

TigerStopAPI/TigerStop\_Com.cs

## TigerStopAPI.TigerStop\_Com.LastAck Struct Reference

### Public Member Functions

* **LastAck** (string ack, DateTime time)

### Properties

* string **Acknowledgement** [get, set]
* DateTime **TimeRecieved** [get, set]

### Private Attributes

* string **acknowledgement**
* DateTime **timerecieved**

### Detailed Description

A struct with fields to track the last acknowledgment received from the machine by the system and the time it was received. Accessible for those functions trying to determine what action has been completed by the machine and performing new actions depending on the response from the machine. The acknowledgment is tracked as a ‘string’ because **TigerStop\_Com.SerialPort\_DataReceived()** retrieves serial responses as ‘string’s.

The documentation for this struct was generated from the following file:

TigerStopAPI/TigerStop\_Com.cs

## TigerStopAPI.TigerStop\_Com.LastCommand Struct Reference

### Public Member Functions

* **LastCommand** (byte[] comm, DateTime time)

### Properties

* byte [] **Command** [get, set]
* DateTime **TimeSent** [get, set]

### Private Attributes

* byte [] **command**
* DateTime **timesent**

### Detailed Description

A struct with fields to track the last command sent from the system to the machine and the time it was sent. Accessible for those functions that need to re-send the last command or determine if a particular command was sent before continuing. The command itself is kept as a ‘byte[]’ because all commands are send to the machine as ‘byte[]’s.

The documentation for this struct was generated from the following file:

TigerStopAPI/TigerStop\_Com.cs

## TigerStopAPI.MessageEvent Class Reference

Inherits EventArgs.

### Public Member Functions

* **MessageEvent** (string newMessage)
* **MessageEvent** (DateTime newTime)
* **MessageEvent** (double newValue)

### Properties

* string **Message** [get]
* DateTime **Time** [get]
* double **Value** [get]

### Private Attributes

* string **message** = string.Empty
* DateTime **time** = DateTime.MinValue
* double **value** = double.NaN

### Detailed Description

A simpler, more manageable, class of event arguments that is used to pass data between events and event subscribers. The most common types of data sent between events in the SDK are ‘string’, ‘double’, and ‘DateTime’. Generally to compare values and data received by the machine to values and data expected by the system to continue or complete an action.

The documentation for this class was generated from the following file:

TigerStopAPI/MessageEvent.cs

# Index

INDEX